Supporting the Investigation and Planning of Pragmatic Reuse Tasks

Reid Holmes and Robert J. Walker
Department of Computer Science
University of Calgary
Calgary AB Canada T2N 1N4
rtholmes,rwalker@cpsc.ucalgary.ca

Abstract

Software reuse has long been promoted as a means to increase developer productivity; however, reusing source code is difficult in practice and tends to be performed in an ad hoc manner. This is problematic because poor decisions can be made either to attempt an unwise, overly complex reuse task, or to avoid a reuse task that would have saved time and effort. This paper describes a lightweight tool that supports the investigation and planning of pragmatic reuse tasks by developers. The tool helps developers to identify the dependencies in the source code they wish to reuse, and to decide how to deal with those dependencies. Questions about pragmatic reuse are evaluated through a survey of industrial developers. The tool is evaluated through the planning and execution of reuse tasks by industrial developers.

1. Introduction

As developers write code, they encounter situations where the feature they are developing is in some way familiar to them; either they have developed the same feature before, or they know of some existing software that has it. Such situations entail pragmatic software reuse, an effective way to reuse source code when applied by experienced industrial developers. However, Krueger states that, "In practice, the overall effectiveness of [pragmatic software reuse] is severely restricted by its informality" [8]. Before such a feature can be reused, the extent of the feature, and the scope of its dependencies upon its system, must be understood [5]. Without this information, a developer cannot make an informed decision about whether or not to proceed with the reuse task. To collect this information, the developer navigates through the system's source files, a process that requires them to manually locate and evaluate a large collection of facts about the dependencies in the source code, and to remember the details of each small-scale decision they have made.

To make reasonable decisions about whether and how to proceed with a pragmatic reuse task, the developer must make a series of smaller-scale decisions about the functionality they wish to reuse. In which classes or methods is it implemented? What dependencies do those classes and methods have? What should be done about those dependencies?

We have developed a lightweight tool, called Gilligan, that supports the systematic investigation of these questions and records the decisions made by the developer. In this way, the developer can devise a well-considered plan and address their final, key question: Is the effort needed to enact this plan worthwhile? By reducing the informality of pragmatic reuse tasks we hypothesise that our tool can help developers to effectively plan larger, more complex reuse tasks.

To evaluate our hypothesis, we performed two separate evaluation steps. First, we conducted a survey of 12 industrial developers working at different organizations to investigate the premise that industrial developers actually perform pragmatic reuse tasks. The survey also investigated how developers think about reusing source code and what other issues arise during the pragmatic reuse process. We then gave Gilligan to four industrial developers and asked them to use the tool to plan reuse tasks they encountered in their workplace. These developers applied the tool to reuse tasks of various sizes, carried the tasks to completion, and provided us with comments about Gilligan's effectiveness.

The remainder of this paper is structured as follows. Section 2 provides a sample scenario to illustrate the nature of the reuse tasks we aim to help developers perform. We describe our tool in Section 3. Related work is described in Section 4. Section 5 details how we validated our approach while Section 6 discusses our results, outstanding issues, and future work.

This paper contributes a model for investigating and planning pragmatic reuse tasks and a tool for interacting with, and reasoning about this model. An initial industrial evaluation of this tool is also described.

2. Scenario

Consider a pragmatic reuse task involving a tool (called UltiGPX) for visualizing Global Positioning System (GPS) data collected by hikers during an excursion. UltiGPX provides a simple visualization of the latitude/longitude coordinates of the hikers' route (see Figure 1, upper right). No display of the changes in elevation (an "elevation profile") is provided by UltiGPX, however. The developer considers such information to be useful to his intended users. The developer has encountered a visualization, within another system, the Azureus BitTorrent client, that seems appropriate for his intent.

Azureus is an open source Java BitTorrent client that downloads files from a peer-to-peer network. Azureus is a complex program that provides numerous visual widgets to help display the download progress of the files that users are acquiring. Specifically, the tool provides a line graph that visualizes network bandwidth.

The UltiGPX developer realizes that this graph is visually similar to what he requires. The panel at the bottom of Figure 1 shows the Azureus network visualization widget superimposed on UltiGPX. However, the goal of Azureus is to support the downloading of files, not to provide reusable APIs for its visual widgets. Visually, the Azureus feature looks exactly right; however, it seems unlikely that a feature providing real-time network visualization would be appropriate to use within a static GPS-data viewing application.

The developer wants more than a high-level intuition whether to pursue this task or not; he wants to know how dependent the graph visualization feature is on the rest of Azureus. In order to do this, he investigates the source code manually within an integrated development environment (IDE). First, he searches for some part of Azureus involved in network visualization; this quickly leads him to the `org.gudy.azureus2.ui.swt.components.graphics` package, in which `SpeedGraphic` seems like the most relevant class. The developer starts by scrolling through the 322-line class, trying to identify which lines are are useful to reuse and which are not.

The developer starts with the `drawChart(..)` method, as this sounds most relevant to the task he is considering. To investigate the implications of each dependency in this 82-line method, the developer must examine each statement to determine which types are referenced. He then needs to look at each type to determine its dependencies and to decide whether or not to reuse those types in addition to `SpeedGraphic`. In the `drawChart(..)` method, 14 different types are referenced. After navigating through 14 different types, he determines that 7 of the types are common to both UltiGPX and Azureus (they both use the SWT framework) which means these dependencies are already satisfied within UltiGPX. However, for the 7

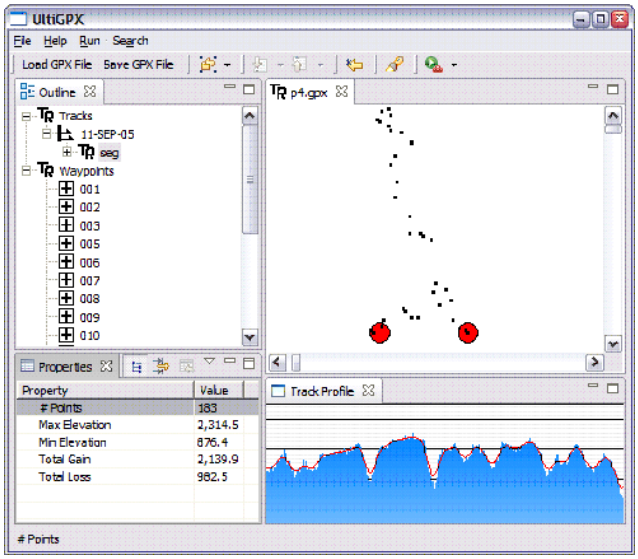

Figure 1. UltiGPX (profile superimposed).

remaining types, the developer must look more critically to determine what to do about their dependencies. Two calls are made to the `AEMonitor` class, both `enter()` and `exit()`. After looking at the `AEMonitor` class, the developer realizes that it provides synchronization functionality within the core of Azureus; this is not necessary within UltiGPX and the developer decides not to reuse the type and to eliminate all references to it within the reused code. Similar situations arise for `COConfigurationManager` and `ParamaterListener` which are involved with the the Azureus preferences architecture. The developer does decide to maintain the dependencies on `Scale` as well as its super types `ScaledGraphic` and `BackgroundGraphic`. The developer also notes that the `Colors` class in Azureus closely corresponds to a class within UltiGPX and decides that while he will not reuse `Colors`, he will remap references to it to his own class.

The developer has now investigated 15 different source files and made decisions about the importance of each of them (including specific decisions about portions of `SpeedGraphic`). While he cannot enumerate the decisions he has made, he has a sense that the task should be manageable and begins to actually carry out the task. He now copies those classes that he has deemed relevant from Azureus into UltiGPX. Next he manages the dependencies based on the decisions he has made, functionality that was rejected is either commented out or stubbed out. References to `Colors` are updated to the appropriate UltiGPX class. While carrying out the task seems simple, it is difficult for the developer to remember all of the decisions he made while navigating between these various source files. Indeed, when he started to do the task, he had to revisit several files to remember what decisions he had made. Additionally, the developer

never actually knows if he is "done" investigating the code; he may have missed an important dependency when he was navigating the various files and may not find out about it until he actually attempts the reuse task.

Reusing the widget from Azureus involved identifying, delineating, and extracting the widget from Azureus and integrating it into UltiGPX. This was not trivial due to the size and complexity of the Azureus project and the fact that the original developers made no effort to make their widgets reusable. The task was complicated by the fact that several different Azureus classes were involved in the final feature, making it difficult to create a clear mental model of the reuse task. These problems would be overcome if the developer could more easily determine, analyze, and record their decisions about the existence, and extent of, the dependencies within the features he wanted to reuse.

3. Gilligan: Supporting reuse

To overcome the problems that arise from situations like our sample scenario, we have developed Gilligan, a plug-in for the Eclipse integrated development environment (IDE)[1], that supports pragmatic reuse tasks. Gilligan helps developers to create a reuse plan by navigating and annotating the structural dependencies of any piece of source code. This reuse plan describes how difficult it will be to extract a feature from its existing system; the plan also gives the developer a sense for how well the feature might fit into his system.

By helping developers record their decisions while investigating features, Gilligan systematizes the feature investigation process; Figure 2 demonstrates the process our tool supports. The developer first notices the task he is undertaking is similar to something he has either done, or encountered, in the past. He then locates the project containing the feature he remembers, and wants to reuse, and starts the Gilligan tool. He then selects a starting point within the project to begin investigating. The developer is then presented with a series of views through which he can navigate the structural dependencies within the feature. While navigating these dependencies, he annotates his decisions about the relevance of the dependencies of the feature he is trying to reuse, converging on a reuse plan. If the task seems infeasible the developer can decide not to pursue the reuse plan at any further; however, if the plan is reasonable the developer can use it to guide his actions as he reuses the feature.

This section describes how to begin a pragmatic reuse investigation (Section 3.1), how to investigate (Section 3.2), and triage (Section 3.3), the structural dependencies in the code, and how to evaluate the reuse plan (Section 3.4). Section 3.5 provides a description of how Gilligan would be applied to the scenario from Section 2. Figure 3 shows the complete Gilligan tool; this figure will be referred to throughout this section.

[1] http://eclipse.org

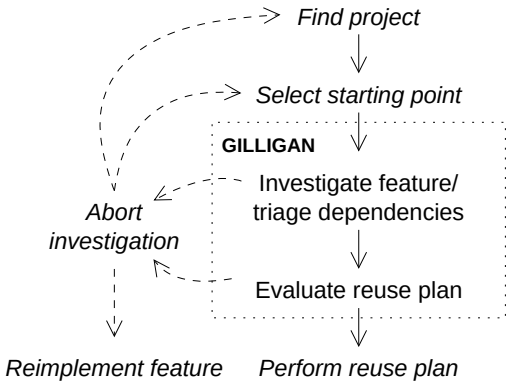

Figure 2. Pragmatic reuse process.

3.1. Beginning a Pragmatic Reuse Task

A developer starts Gilligan by selecting the project containing the feature he wants to reuse (e.g., Azureus) and the project he wants to reuse the feature within (e.g., UltiGPX). Gilligan statically analyzes the the source code from these projects and extracts their structural dependencies. Both projects are identified in advance so the classes common to both can be identified. The developer is then presented with a searchable tree of the source system from which he selects a starting point for the reuse task: a package, class, or method that he thinks is involved in the feature he wants to reuse. The developer is then presented with a visual view populated with this starting node (as in Figure 3, top right) that he can begin to investigate.

3.2. Navigating Structural Dependencies

Using the visual view, a developer can investigate the structural dependencies within a feature. This view provides an abstraction of the structural dependencies within the feature using a graph representation; nodes represent packages, classes, methods, and fields; edges between the nodes represent structural relationships (inherited-from, calls, references, declared-by, contained-by). By providing an abstraction of the reuse task, the developer can visually see which nodes he has visited in order to systematically investigate those he has not. It also helps him to focus on the actual dependencies rather than expending effort navigating source files to identify those dependencies manually.

The developer can double-click on a node to view its structural dependencies. The node's structural dependencies are then shown on the graph: edges are either added to other, already visible, nodes or new nodes may appear on the graph with smooth animated transitions (to minimize distraction and maintain context). When a node is selected, it becomes highlighted as do its edges and those nodes that

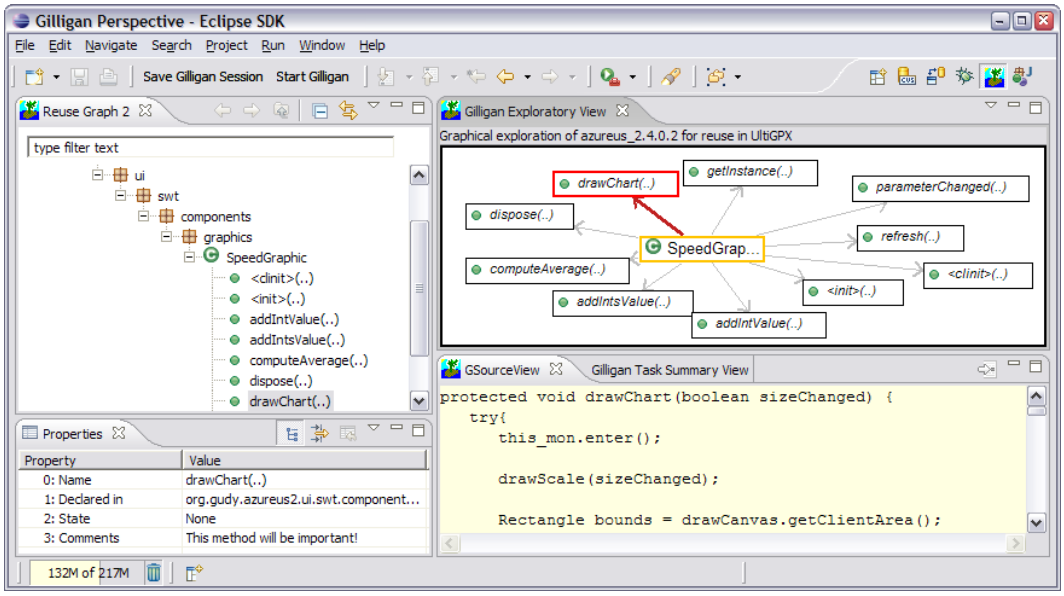

Figure 3. Screen shot of the Gilligan Tool.

are structurally related to it. This allows the developer to quickly see those nodes upon which a particular node is dependent and the edges representing those relationships.

To help manage the complexity of the graph, nodes can be collapsed into their parents. This collapse functionality simplifies the graph by eliding details the developer is no longer interested in seeing (such as collapsing methods into its parent class, or a class into its package). In Figure 4, the developer has collapsed many nodes into the `swt` package. By looking at the tree view (Figure 3, left) the developer can see that this collapsed node represents 21 other nodes (2 packages, 5 classes, and 12 methods).

The developer can also request the source code for any node or edge in the graph (except for package nodes and contains edges). Gilligan provides the most specific amount of information possible for any source request (e.g., source requests for method nodes display only the code for that method). If the developer requests the source for an edge, they are presented with an annotated source view. For example, for a calls edge, the source for the method in which the call is made is highlighted with the specific statements pertaining to that call. By choosing to see the source for particular edges the developer can quickly determine which portions of the source are involved with any given structural relationship; this helps him to focus on the relationship he is interested in without getting distracted by other structural relationships (some of which he may have already triaged) in the source code.

3.3. Triaging Structural Dependencies

While the developer is navigating the structural dependencies, he is making decisions about each node's applicability to his reuse task. Gilligan provides a lightweight way for the developer to record his decisions about these nodes; he can simply click on the colour tool that corresponds to his decision for that node. The decisions he makes are recorded by the colour of the node; these colours allow the developer to quickly get a sense for the reuse task by just glancing at the visual view. Nodes that have yet to be decided upon are shown in plain white. In addition to the broad annotations described below, the developer can also annotate any node with arbitrary text that records any thoughts or special instructions. These text annotations are entered and viewed in the properties view (Figure 3, bottom left) for the node. Nodes can be annotated in four ways (in addition to the textual annotations):

Accept. By accepting a dependency, the developer is acknowledging that they wish to reuse the source code the node corresponds to. This means that the developer intends to move the source code from its current context into their own project. This decision is indicated by the colour green in our visualization.

Reject. Unwanted dependencies are those that provide functionality that the developer does not want to be reused. In this case, the developer knows that he does not want these references in his system and will not reuse them (or analogous functionality) with the accepted nodes. When the reuse task is being performed, references from accepted to rejected nodes must be dealt with by the developer (they are frequently just commented out). This decision is indicated by the colour red in our visualization.

Remap. This decision means that, while the functionality is needed (for a node that has been accepted to function correctly), the developer wishes to re-target it to an already existing piece of code in their own system that can provide

the required service. This decision is indicated by the colour blue in our visualization.

Already Provided. Our tool automatically colours those nodes that are common between both the source system and the target system. For example, for any Java project, references to `java.*` packages are not dependencies that need to be migrated. Such nodes are indicated by the colour yellow in our visualization.

These annotations are of key importance to the reuse plan. By promoting the developer's decisions to the forefront with bold colours, the developer can, with a glance, get a sense for the number of dependencies in the graph to be handled through each kind of treatment and thus the scope of the challenge facing them, should they choose to pursue the reuse task.

3.4. Evaluating the Reuse Plan

Gilligan helps the developer focus on those dependencies that matter, recording his decisions about those dependencies to minimize the re-viewing of code fragments. Using the graphical view, it is visually evident which nodes need to be further addressed before the investigation is complete. While this systematic process helps the developer to see what decisions he has made, it does not impose any particular order in which those decisions must be made; the developer can iterate on the graph in any way that is appropriate to his task.

At any point during the investigation of the feature, the developer can evaluate the state of their reuse plan. The developer may notice early on that there are far too many rejected and remapped dependencies to easily reuse the feature. If the developer completes the reuse plan, they can use its structural description and annotations to make an informed decisions about reusing the feature.

3.5. Application to the sample scenario

To ensure that Gilligan can successfully plan a reuse task we applied it to the scenario described in Section 2. We started by selecting Azureus [2] as the origin of the feature we want to reuse and UltiGPX as the project for the code to be reused within. After textually searching the source code, we find the `SpeedGraphic` class, which seems like a reasonable place to start the investigation (the text references both both network speed and graphics). Gilligan's initial visual state is shown in Figure 3. We navigate the `SpeedGraphic` class and its dependencies to identify those portions of the class that are relevant to the graph drawing feature we want to reuse while identifying any other Azureus dependencies we do not want to reuse.

Opening the dependencies for the `drawChart(..)` method, we find 21 structural dependencies; however, 14 of these (from 4 classes) have been automatically coloured yellow by Gilligan—both Azureus and UltiGPX use the SWT framework. To reduce the clutter on the screen, we collapse all of the `org.eclipse.swt.*` dependencies into a single package. The remaining 7 dependencies are from four different classes within Azureus. Double-clicking on the call edge to `AEMonitor.enter()`, we are presented with `drawChart(..)`'s source code highlighted with the references to `AEMonitor`. From this annotated source view we can see that `AEMonitor` is concerned with locking within the core of Azureus. This is not a feature we care to reuse so we collapse `AEMonitor`'s methods into itself and mark it as rejected. Another 15 minutes of investigation results in the reuse plan shown in Figure 4.

Using Figure 4, we can see that we are going to reuse 5 complete classes and 1 partial class. We are going to have to manage 4 dependencies on source code (involving the Azureus network locks and configuration) that was rejected from the reuse task. Five classes in the `swt` package were common between Azureus and UltiGPX; no action is required of the developer to satisfy dependencies on these classes. From this sketch, the we can see that it should not be too difficult to extract this feature and integrate it with UltiGPX. Performing the extraction and integration—and following the plan—took less than an hour and resulted in 708 lines of reused code; in this process we had to comment out 2 methods and 5 other lines of code to conform to the decisions we made about rejected dependencies. The reuse plan helped in accomplishing the task as whenever a compile error was encountered in the reused code fragments we could check the reuse plan to see how we should manage it, this helped direct our integration plans.

It is important to note that we could not have simply called graphic functionality in Azureus as they did not provide an API for this feature. This is not unreasonable as they could not have foreseen reusing this functionality within a GPS application. While we could have written this code from scratch, by reusing the functionality we were able to leverage the feature they had created—and tested—over several versions of their own product. The reuse plan provided by Gilligan helped to show what (and how many) dependencies would be involved in the reuse task, allowing us to make an informed decision about the feasibility of the task.

4. Related work

Reuse has long been investigated in the literature as a solution for many problems plaguing software engineers [10]. However, reuse research has focused primarily on the creation of reusable software components and libraries and using these to create software end-products. While pragmatic reuse tasks (also called code scavenging) have been shown to be effective [8], little work has been done to follow-up on these findings.

[2] http://azureus.sf.net (v2.4.0.2)

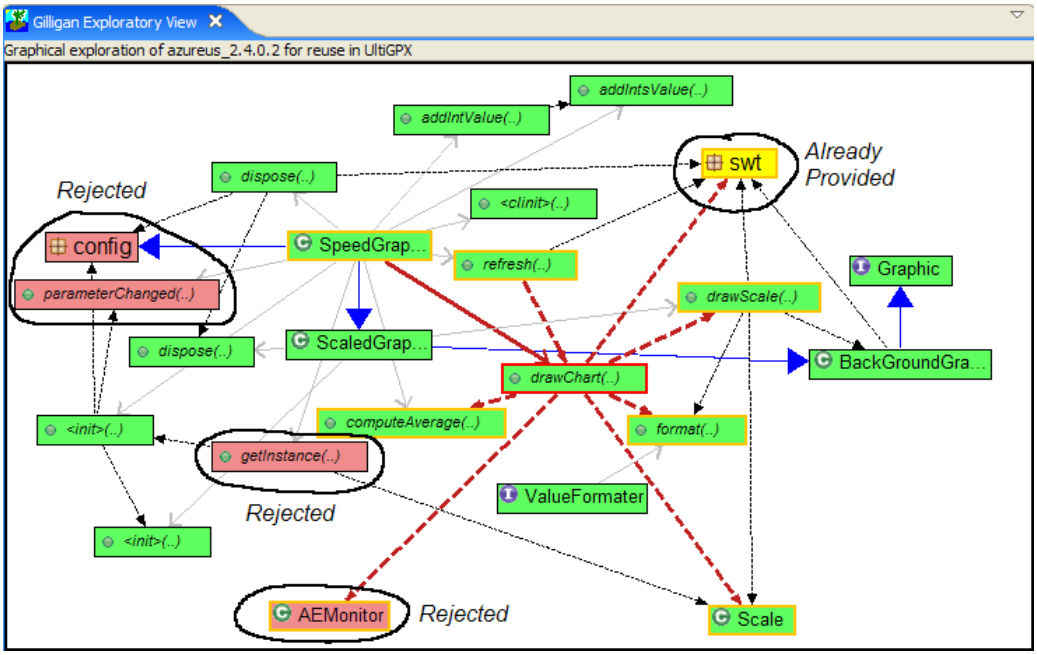

Figure 4. Azureus graph feature reuse plan. (Screenshot is annotated for greyscale reproduction.)

4.1. Past approaches

While many approaches have advocated refactoring code into reusable application programming interfaces (APIs), this is not always possible. The original code may no longer be maintained or its maintainers may not be willing to refactor the code to meet the new requirements. Indeed it has been shown that reused code must be frequently modified in some way to work within its new context [14]. Frakes and Kang note that dedicated reuse strategies within companies require a large up-front cost that must be justified in terms of business goals. They also found that most software systems are variants on pre-existing systems [4]. As new systems are extensions of the old, it is natural that pragmatic reuse will take place in situations where the new requirements do not align perfectly with the old.

Our tool addresses two issues identified by Frakes [3] through an industrial reuse questionnaire. First, he identified that CASE tools may not be effective at promoting reuse. By extending a popular IDE with features specific to pragmatic reuse tasks, we hope this can improve the environment's ability to help developers engage in these tasks. Second, he identified that the lack of process hampers reuse efforts. Our tool provides developers with a unified approach for investigating reuse tasks; this approach can help reduce the ad hoc nature of source traversal inherent in how developers currently perform unanticipated reuse tasks.

4.2. Developer practices

Parsons and Saunders [13] determined that developers were able to perform tasks by anchoring their understanding to existing code and adjusting the code to meet their needs. While this evaluation was only tested for one small case (albeit with many developers), it is an encouraging endorsement for so-called "white box" reuse. By providing developers with a concrete reuse plan, we aim to help developers anchor the reuse task so they can better understand how the code needs to be adjusted to meet their needs.

Selby [14] analyzed 25 projects at NASA and discovered that 32% of the modules within those projects were reused from prior projects. Of these reused modules, 47% required modification from their original form, while 53% required no modification. This study showed that a significant proportion reuse tasks involve source code modification; we take this as further motivation for our approach which aims to support developers in this goal.

Reuse in the manner we are advocating can be seen as creating code clones. While these clones have in the past been perceived negatively, new research has found that clones are frequently short lived, and when they are long-lived they are not easily refactored [7]. Short-lived clones are those that are reused and then modified to meet the new system's requirements. Non-refactorable clones indicate that the original API could not be refactored to meet the requirements of both the old and new usage. These cases are no worse than implementing the features from scratch but the developers still get the added benefit of having reused code. One problem with reusing code in this manner, however, is that when bugs are fixed in the original source they are not automatically propagated to the reused versions; support for this type of process is an active research topic [6].

4.3. Feature location

Feature location approaches help developers locate those portions of the source code relevant to a particular feature. Chen [2] investigated feature location using a graph-based technique, although this approach was purely a generic program understanding tool. The FEAT [12] tool helps developers create concerns, descriptions of scattered software. Our approach extends this model by providing annotations for developer decisions and specifically targeting it at pragmatic reuse tasks. Recent work by Robillard [11] suggests that Gilligan could be further enhanced by automatically recommending nodes of interest to the developer.

Other systems have looked at creating reusable components from existing code. CARE [1] takes a metrics-based approach to identifying reusable components while Lanubile and Vissagio used program slicing to identify these components [9]. While these tools can identify many components that are reusable, our goal is to help the developer prepare a reuse plan for the specific feature they wish to reuse.

Our research focus is different from these previous approaches. We are interested in how industrial developers undertake pragmatic reuse tasks. Instead of recommending these developers create and use reusable components, we are interested in working within the framework of their practices by helping them accomplish pragmatic reuse tasks better.

5. Evaluation

Our evaluation consists of two parts. First, we conducted a survey with with industrial developers into their reuse practices. Second, we gave Gilligan to several industrial developers and asked them to use it to plan their reuse tasks. These evaluations were poised to answer two key questions: Do developers perform pragmatic reuse tasks?; Can Gilligan help developers plan industrial reuse tasks?

5.1. Reuse in industry

This work depends on one important premise: developers engage in pragmatic reuse tasks. In order to investigate this claim we conducted a survey with twelve developers from six companies. The subjects had between two and twelve years experience working in industry. Each of the companies produce software targeted at different domains. Our respondents were all employed in full-time active development positions. This survey investigated how developers think about, and perform, software reuse tasks through a set of statements (which subjects were asked to agree/disagree with on a seven-segment Likert scale) and a set of open-ended questions. The key statements and responses are summarized in Table 1. From these surveys we have identified three main themes:

Developers are performing pragmatic reuse tasks. Developers agreed that they had reused source code (Table 1, # 1) and that these reuse tasks frequently encompassed whole classes (Table 1, # 2). However, they were split when asked about reusing whole features (Table 1, # 3). In the long answer section the developers indicated that their reuse tasks usually ranged from 4 lines to 50 (several methods or a portion of a class), but sometimes included whole classes (up to 1000 lines). Their comments also indicated that reuse of this nature frequently occurred while prototyping new features, or in the early stages of a project when functionality was incorporated from existing products. Developers generally agreed they would rather reuse a feature than reimplement it themselves (Table 1, # 4).

Developers reuse code to save time and improve quality. The most popular reason for reusing source code was so the developers could save themselves time (Table 1, # 5). This was backed up repeatedly in the written questions with comments such as "reusing code is quicker and easier than [starting from scratch]". The next major reason for these reuse activities was to increase the reliability of their code (Table 1, # 6). The developers wanted to "leverage existing testing". The desirability of a piece of code was increased if tests were available as it increased the developers trust in the quality of a piece of code.

Developers want to understand a feature's dependencies. Reasoning about source code, especially code someone else has written, can be very difficult. Our subjects agreed (Table 1, # 7) that keeping track of the facts relevant to a reuse task while navigating the source code was difficult. Specifically, identifying the dependencies of the code they wanted to reuse on the system in which it was designed was of importance (Table 1, # 8).

While attempting to understand a particular piece of code, many developers sketched out its structure visually on paper. Several others wrote notes either on paper or as annotations within the code itself. Other developers still would copy the code out of its original context and into their environment to see how the code "bleeds" (compilation errors are shown in red in their IDE) in order to get a feel for how compatible the code fragment might be with their system.

Developers have access to large amounts of code. Our respondents strongly agreed that their organizations had large repositories of code available to reuse from (Table 1, # 9). Additionally, they reported that portions of the features they developed were available in other systems for which they had access to the source code (Table 1, # 10).

5.2. Industrial evaluation

To ensure developers could plan, and perform, real reuse tasks using Gilligan, we gave the tool to four industrial developers working in different companies. We identified these four developers during our survey as they all work

#	Question	Strongly Agree	Agree	Somewhat Agree	Neutral	Somewhat Disagree	Disagree	Strongly Disagree
1	I have reused source code	10	2	0	0	0	0	0
2	I have reused whole classes	7	3	2	0	0	0	0
3	I have reused whole features	4	0	1	1	4	2	0
4	I would rather reimplement a feature than reuse an existing one	0	0	2	1	3	4	2
5	I reuse code to save time	6	4	2	0	0	0	0
6	I reuse code to increase reliability	4	7	0	1	0	0	0
7	Keeping track of the relevant details of a piece of source code while navigating its text can be difficult	2	6	2	1	0	1	0
8	Understanding what dependencies a feature has on its context is important for me to determine whether I should reuse it	7	5	0	0	0	0	0
9	My organization has a large amount of code available to be reused	5	1	4	1	1	0	0
10	Portions of features I am developing already exist	1	4	5	0	1	1	0

Table 1. Responses from the industrial reuse questionnaire.

with the the Java programming language within the Eclipse IDE, the same environment Gilligan currently supports. In addition to verifying that the developers could plan reuse tasks, we wanted to know if the developers felt they could tackle larger reuse tasks with the tool than they would normally attempt. The four developers applied our tool to their reuse tasks and filled in a short questionnaire about each task they tried. Gilligan was also instrumented to note how they interacted with the tool while they investigated their tasks.

5.2.1 Case study 1

The first developer undertook two tasks: he extracted code from the open-source SWT framework for parsing both BMP and PNG image files. He wanted to reuse these pieces of code because they involved complex binary file format I/O that he did not want to have to write himself and he was unable to reuse all of SWT (which comprises 68 KLOC spread across 458 classes)[3].

BMP extraction. The developer started Gilligan with the `WinBMPFileFormat` class as the initial node. Using the graphical view, the developer was able to quickly annotate several methods in this class as rejected as they pertained to the writing of these files, which he was not interested in. Further exploration led him towards `LEDataInputStream` and `ImageData`. He reused the former in its entirety, and just the data structure from the latter. In the end, he reused 497 lines of code and had no latent dependencies on SWT. Of the 14,081 possible nodes in SWT, the developer only visited 60. His final view of the feature had 27 visible nodes. Of the nodes he visited, he accepted 38 of them, rejected 16, remapped 2, and 4 were already provided.

The developer then demonstrated how he would have undertaken the task manually. First, he copied `WinBMPFileFormat` into his new project. He then went down the list of compilation problems (there were many) and dealt with them individually. Any dependency he could not easily manage he left until later. At the end he went through the remaining difficult dependencies and also copied `LEDataInputStream` and the `ImageData` data structure into his workspace. Once the compilation errors were resolved he was done. His methodology was similar to what our tool provides: he used the compilation errors as markers for structural dependencies that were not satisfied within his target environment. Unfortunately, doing this manually forces the developer to encounter hundreds of these errors at a time which can be difficult to manage.

PNG extraction. The developer began his investigation with the `PNGFileFormat` class. He was interested in immediately noting all of the class-level dependencies of this 471-line class. Unfortunately, the tool is currently designed to support a more bottom-up investigation style and he had to open `PNGFileFormat`'s methods to see these dependencies. After opening these dependencies he had 92 nodes on the screen and had discovered that there were at least 20 classes of interest to him for this task. During this investigation the developer was interrupted multiple times by co-worker questions. After these interruptions he was able to go back to the visual plan to remember where he was; because his decisions were noted on the plan, these

[3] http://www.eclipse.org/swt/

distractions did not cause him to go back and re-evaluate any nodes. With 92 nodes visible, the developer indicated he would appreciate being able to filter the nodes based on their type (for instance, only show class nodes) in order to make it easier to understand. In the end, the developer marked 20 classes as accepted, 2 as rejected and 1 as remapped.

When the developer actually did this reuse task he reused 23 classes (comprising $\tilde{3}$000 LOC). During this reuse task he decided to change his mind about two decisions, he accepted one previously-rejected class, he changed the remapped class to accepted, and accepted one class he overlooked during the investigation. These changes were made primarily due to the complexity of the task he was pursuing.

In the post-task questionnaire the developer stated, "After trying to reuse the SWT BMP decoder, I wasn't convinced additional tooling was necessary (300 lines reuse, fairly isolated to one to three classes). After trying to reuse the SWT PNG decoder, I changed my mind." He also indicated that he strongly agreed that Gilligan could help him "attempt larger, more complex reuse tasks". However, this case highlighted the need for further refinement of the user interface. While Gilligan did initially help the developer identify those types that were relevant to his task, he was eventually overwhelmed and had to use a hybrid approach that used both Gilligan and the manual techniques demonstrated in the last case. In Section 6 we discuss future work to address his concerns.

5.2.2 Case study 2

This developer wanted to reuse a feature that serialized some of his objects in Java into XML so they could be sent over the wire. This feature needed to be reused because the originating project was no longer being maintained. The developer started with a class he knew was involved with the XML serialization functionality and explored its dependencies. He accepted 4 classes, remapped 2, and found that 8 were already provided within his target system. During his investigation he investigated 84 nodes (33 of which were visible in the final plan), accepting 13, rejecting, 2, remapping 4, while 18 were automatically marked as common. This reuse task took approximately 2 hours and in the end $\tilde{9}$00 lines of code were reused. The reuse task was actually a starting point for a refactoring task to make the old feature conform to the new system. The developer found that the tool "helped me visualize the scope of reuse tasks and how much I would be able to reuse and what I would have to write."

5.2.3 Case study 3

The third developer wanted to reuse the virtual file system from a 3rd party application to add this functionality to his own system. The reuse task involved reusing 9 classes and remapping two of the classes to equivalent functionality already provided within his own system; this task involved reusing $\tilde{3}$000 lines of code. The developer investigated 49 nodes in the visual view, with 32 of them remaining in his final reuse plan; this investigation took 25 minutes. He accepted 25 nodes, while 5 were already provided. He also marked two nodes for remapping; he wanted to connect these nodes, corresponding to logging functionality, to those within his own application. During his investigation, he also investigated the source code for 12 specific method calls. This developer found that Gilligan helped him decide that this task was possible before carrying it out; however, he wanted the tool to clearly highlight which method-nodes had external dependencies on them. He also wanted the ability to hide nodes such as those representing already provided functionality.

5.2.4 Case study 4

The fourth developer also completed two tasks. In his first task, he wanted to reuse an implementation of an old feature within a new system; however, he needed to modify this implementation in ways that were inconsistent with the old application. In this task, he visited 42 nodes, with 24 ultimately being of interest to him. Of these, he marked 12 as accepted and 5 as remapped; 15 were automatically marked as already provided within his target system. The task ultimately reused only 200 lines of code and took only 20 minutes to accomplish.

In the second task, the developer attempted to reuse the GraphML parsing code from the Jung open-source project[4]. This was a complex task that involved over 2000 lines of reused code. During the investigating phase, the developer identified the need for a "verify plan" feature for the tool; this feature would check one's accepted nodes and confirm that they do not have any non-triaged dependencies. He requested this feature because during this task he investigated 72 nodes (of the 23,157 nodes in Jung) which he found to be somewhat overwhelming to keep track of. He noted that the tree view was especially important for tracking large reuse tasks as the entries in this view are also coloured with the developer's decisions. In this task he accepted 10 nodes, rejected 4, remapped 5, and 13 were marked as already provided.

6. Discussion

Usability improvements. In the case studies (Section 5.2), the developers made some specific suggestions to improve Gilligan's user interface. These suggestions are primarily aimed at helping the tool to scale to larger reuse tasks, these include implementing node filtering and ranking features that can help elide, and promote, nodes from the graph. Developers also wanted the ability to have the

[4] http://jung.sf.net

tool check to ensure the reuse plan was complete; we are currently implementing these improvements.

Semi-automating the reuse task. Planning pragmatic reuse tasks is only the first step to the pragmatic reuse process; we are currently investigating providing support for the semi-automatic application of the reuse plan. Using a complete reuse plan the tool can extract the necessary code and help the developer apply program transformations to manage some of the remapped and refused dependencies.

Evaluation summary and improvements. In our evaluation we sought to confirm that: (1) industrial developers perform pragmatic reuse tasks; and (2) these developers can successfully plan reuse tasks with Gilligan. Our survey confirmed that developers do perform these reuse tasks and they carry them out manually using standard IDE tools. Our case studies show that industrial developers can successfully plan, evaluate, and carry out pragmatic reuse tasks with Gilligan.

With the planned extensions to Gilligan, a more through evaluation is required. An experiment to compare the effectiveness of developers supported by our tool, compared to their standard practice, is needed to ensure that our technique can help improve developer productivity and reduce error-rates. As this type of evaluation is difficult and time-consuming to conduct well, we plan to extend Gilligan before this takes place.

In the case studies performed to date, all four developers agreed that Gilligan helped them plan larger reuse tasks than they would usually attempt; indeed, the tasks the developers attempted were all larger than the typical sizes identified in the survey. These case studies involved four industrial developers who performed six realistic, pragmatic reuse tasks. While we cannot claim their experiences will definitely generalize, these developers, and their tasks, were varied enough to suggest that Gilligan is a valuable tool for planning, and reasoning about, pragmatic reuse tasks.

7. Conclusion

Developers must undertake pragmatic reuse tasks in situations when traditional component-based reuse is not feasible. The literature, along with our industrial survey, confirm that these types of reuse tasks frequently arise in industry. We have developed a lightweight process to help developers plan pragmatic software reuse tasks. We performed several case studies, with industrial developers, using the the tool we created to realize this process. These developers confirmed that the process our tool promotes aligns with how they perform pragmatic reuse tasks; the developers successfully used the tool to plan larger reuse tasks than they would typically attempt. The developers were then able to perform their reuse tasks based on these plans. Future enhancements, particularly those supporting the semi-automated application of these reuse plans, will lay the foundation for a more through experiment investigating the effectiveness of our approaches compared to current developer practice. While our current results remain exploratory, they are an encouraging endorsement of our proof-of-concept tool, and process, in an industrial setting.

8. Acknowledgments

We would like to thank the subjects who responded to our surveys and participated in our case studies. This research was funded in part by IBM and in part by the Natural Sciences and Engineering Research Council.

References

[1] G. Caldiera and V. R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70, 1991.

[2] K. Chen and V. Rajlich. Case study of feature location using dependence graph. *Proc. Int'l Wkshp. Program Comprehension*, pages 241–247, 2000.

[3] W. B. Frakes and C. J. Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–87, 1995.

[4] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31(7):529–536, 2005.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12(6):17–26, 1995.

[6] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proc. Int'l Wrkshp. on Mining Softw. Repositories*, pages 58–64, New York, NY, USA, 2006. ACM Press.

[7] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. Joint Europ. Softw. Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pages 187–196, 2005.

[8] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[9] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. on Softw. Eng.*, 23(4):246–259, 1997.

[10] D. Mcilroy. Mass-produced software components. In *Softw. Eng., Report on a conf. sponsored by the NATO Science Committee*, pages 88–98, 1968.

[11] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proc. Joint Europ. Softw. Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, pages 11–20, 2005.

[12] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. Int'l Conf. Softw. Eng.*, pages 406–416, 2002.

[13] C. Saunders and J. Parsons. Cognitive heuristics in software engineering: Applying and extending anchoring and adjustment to artifact reuse. *IEEE Trans. Softw. Eng.*, 30(12):873–888, 2004.

[14] R. W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Trans. Softw. Eng.*, 31(6):495–510, 2005.